Trait-oriented Programming in Java 8 *

Viviana Bono Enrico Mensa Marco Naddeo
University of Torino, Italy
{bono,naddeo}@di.unito.it, enrico.mensa@gmail.com

Abstract

Java 8 was released recently. Along with lambda expressions, a new language construct is introduced: default methods in interfaces. The intent of this feature is to allow interfaces to be extended over time preserving backward compatibility. In this paper, we show a possible, different use of these interfaces: we introduce a trait-oriented programming style based on an interface-as-trait idea, with the aim of improving code modularity. Starting from the most common operators on traits, we introduce some programming patterns mimicking such operators and discuss this approach.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.2 [*Language Classifications*]: Object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.13 [*Reusable Software*]: Reuse models

General Terms Design, Languages

Keywords Java 8, Default method, Trait, Programming Pattern, Code Modularity

1. Introduction

From the point of view of the language constructs, the most prominent addition in Java 8 is the *lambda-expression* construct, that comes along with an apparently secondary construct, that is, the *default method* (aka *virtual extension method*, aka *defender method*) in interfaces. The primary intent of this feature is to allow interfaces to be extended over time preserving backward compatibility. These features of Java 8 are described in the proposal *JEP 126* (JDK Enhancement Proposal 126) *Lambda Expressions & Virtual Extensions Methods* [9]. JEP 126 is a follower of the Project Lambda, that corresponds to JSR 335 (Java Specification Request 335) [16].

A *default method* is a virtual method that specifies a concrete implementation within an interface: if any class implementing the interface will override the method, the more specific implementation will be executed. But if the default method is not overridden, then the default implementation in the interface will be executed.

* Partially supported by MIUR PRIN Project CINA Prot. 2010LHT4KM and Ateneo/CSP Project SALT. Authors listed in alphabetical order.

PPPJ '14, September 23 - 26 2014, Cracow, Poland.

http://dx.doi.org/10.1145/2647508.2647520

In previous Java releases, interfaces were to provide multiple type inheritance, in contrast to the class-based single implementation inheritance. Java 8 interfaces, instead, introduce a form of multiple implementation inheritance, too. Therefore, they are similar to *traits* [10], which are sets of methods. [1] Java 8 interfaces, then, can be exploited to introduce a *trait-oriented* programming style. Note that we are not proposing a linguistic extension of Java 8 with traits, but programming patterns within Java 8, with the goal of improving *code modularity* and, therefore, *code reuse*. Starting from operators on traits [10], we introduce some Java 8 programming patterns mimicking such operators and discuss this approach.

The paper is organized as follows: Section 2 illustrates briefly the *trait* construct, Section 3 introduces Java 8 *default methods*, Section 4 proposes the programming patterns inspired by the trait operators, Section 5 makes some comparisons with related work and draws some conclusions.

2. Traits in a nutshell

The possibility of *composition* and *decomposition* of code are important characteristics to care about in a programming language. Let us point out some problems of (single and multiple) inheritance concerning composability:

- *Duplicated features*. Single inheritance is the basic form of inheritance; by means of it we can reuse a whole class (and also add some features); however, if it is necessary to use the features of more than one class, some code must be duplicated.
- *Inappropriate hierarchies*. Instead of duplicating methods in the lower classes, we can bring those methods up in the hierarchy; however, this way we violate the semantics of the upper classes.
- *Conflicting features*. If we have multiple inheritance (as C++ does), a common problem is how to treat conflicts. Method conflicts can be solved (for example, thanks to override), but conflicting attributes are more problematic.

Traits are a possible solution to these problems. A trait is a "*simple conceptual model for structuring object-oriented programs*" [10] and it is a collection of methods. This is very important: traits are statless, they cointain only methods, therefore every conflict of state is avoided. Only method name conflicts must be dealt with, explicitly, by the programmer.

Every trait can define *required methods* and *required fields*. Required fields are indirectly modelled via required setter and getter methods. A trait can be defined directly (by specifying its methods) or by composing one or more traits. The composition is performed by means of the following operators:

- *Symmetric Sum*: a new trait is defined by combining two or more existing traits whose method sets are disjoint. In the case the sets are not disjoint, conflicts arise.

[1] This is pointed out in many places, see, for instance, [17].

- *Override*: a new trait is defined by adding method(s) to an existing trait. If an already present method is added, the old version is overridden. We will refer to this operator as "trait override", to distinguish it from Java override.
- *Exclusion*: a new trait is defined by excluding a method from an existing trait.
- *Alias*: a new trait is defined by adding a second name to a method from an existing trait. This is useful if the original name was excluded after resolving a conflict. Note that, if a recursive method is aliased, the recursive call will be done on the original method.

These operators are from the original proposal [10]. Other operators were introduced in further works; a comprehensive list of operators with relations among them can be found in [6].

The original definition of traits says that trait and class usages are separated: the first ones are units of reuse, while the second ones are generator of instances. A class can be specified by composing a superclass with a set of traits and some *glue methods* (aka *glue code*). Glue methods are written inside a class and make it possibile the connection between different traits. An example of glue code are the setter/getter methods.

Trait composition respects the following three rules [10]:

- Methods defined in a class itself take precedence over methods provided by a trait. This allows glue methods defined in the class to override methods with the same name provided by the traits.
- Flattening property: a non-overridden method in a trait has the same semantics as if it were implemented directly in the class.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Method name conflicts can be resolved directly in classes by adding appropriate glue methods which redefine the conflicting methods, or with trait composition, thanks to the operators:

- with trait override, by adding one method with the same name, which hides the previous implementations and may call whichever of them (through aliases);
- with exclusion, by excluding all but one of the conflicting methods.

Note that the combination of exclusion and alias can be used also to solve conflicts among required methods. In the case of accessor methods, this conflict resolution helps solving field conflicts.

3. On default methods

The role of an interface up to Java 7 was to give a contract to the user (that is, a type), but not to specify any detail of the contract itself (that is, the implementation). The main characteristic of default methods (introduced by a keyword `default`) is that they are virtual like all methods in Java, but they provide a default implementation within an interface.

Java 8 method resolution is defined in [12] and its formalization in a Featherweight-Java style [14] is in [13]. To summarize it, we take the four (informal) rules about method linkage from [13]:

- A method defined in a type takes precedence over methods defined in its supertypes.
- A method declaration (concrete or abstract) inherited from a superclass takes precedence over a default inherited from an interface.
- More specific default-providing interfaces take precedence over less specific ones.
- If we are to link `m()` to a default method from an interface, there must be a unique most specific default-providing interface to link to, otherwise the compiler signals a conflict.

From these dispatch rules, we can extrapolate some examples of behaviour that can help the reader to understand the default method construct.

A first example. If the class that implements the interface using default methods does not override those methods, the default implementation provided in the interface will be executed.

```
interface A {
  default void m()
    {out.println("Hi, I'm interface A");}
}
class B implements A {}
//doesn't override m

public class FirstDM {
  public static void main(String[] args) {
    B b = new B();
    b.m();
  }
}
```

The output will be: `Hi, I'm interface A`.

Classes always win. Class methods have the precedence over default methods:

```
interface A {
  default void m()
    {out.println("Hi, I'm interface A");}
}
class B implements A {
  //overrides m
  public void m()
    {out.println("Hi, I'm class B");}
}
public class SecondDM {
  public static void main(String[] args) {
    B b = new B();
    b.m();
  }
}
```

The output will be: `Hi, I'm class B`.

The most specific interface wins. If no class overrides a default method, the default method with the most specific implementation will be executed:

```
interface A {
  default void m()
    {out.println("Hi I'm interface A");}
}
interface B extends A {
  default void m()
    {out.println("Hi I'm interface B");}
} //more specific because of the 'extends'

class C implements A, B { }

public class ThirdDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

The output will be: `Hi I'm interface B`.

Conflicts are not always avoidable. If a unique most specific default-providing interface is not found, an error will occur:

```
interface A {
  default void m()
    {out.println("Hi I'm interface A");}
}
```

```
interface B {
  default void m()
    {out.println("Hi I'm interface B");}
}

class C implements A, B { }

public class FourthDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

The compiler says:

```
class C inherits unrelated defaults for m() from
types A and B - class C implements A, B { }
```

How to resolve conflicts. The construct `X.super.m()` can be used, where `X` is one of the direct superinterfaces containing the default method `m()`:

```
interface A {
  default void m()
    {out.println("Hi I'm interface A");}
}
interface B {
  default void m()
    {out.println("Hi I'm interface B");}
}
class C implements A, B {
  //calls m in A
  public void m()
    {A.super.m();}
}
public class FifthDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

The output will be: `Hi I'm interface A`.

Note that this new construct is just for resolving conflicts while using default methods and not for a general purpose [12].

About abstract methods. We said that classes always win over interfaces. This is true also when classes are abstract:

```
interface A {
  default void m()
    {out.println("Hi I'm interface A");}
}
abstract class B {
  abstract void m();
}

class C extends B implements A { }

public class SixthDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

The compiler says:

```
C is not abstract and does not override abstract
method m() in B - class C extends B implements A { }
```

This happens because the abstract declaration of `m()` in `B` takes precedence over the default declaration in `A`.

4. A guide to trait-oriented programming

Java 8 interfaces play the role of traits, with default methods as provided methods and abstract methods as required methods. We will refer to an interface with this role with the term "trait" and we introduce the convention that such an interface will be named with a name starting with `T` or `Trait`. As within stateless traits, required fields are encoded as required accessor (getter/setter) methods, that is, as abstract methods, whose implementation will be provided as glue code by the class implementing the traits.

We introduce now the programming patterns matching the trait operators listed in Section 2, then we discuss briefly some drawbacks related to return types in Java overrides.

Symmetric sum. This provides the fundamental feature of multiple inheritance. With "symmetric" it is meant that all the addends of a sum are peers, implying that, in the case of a conflict, it is up to the developer to deal with it. The first example shows a case of a sum without conflicts. We have three traits:

```
public interface TMouth {
  default void makeASound()
    {out.println("Yaaaawn");}
  default void eat(String s)
    {out.println("I'm eating "+s);}
}


public interface TEyes {
  default void lookAround()
    {out.println("I'm looking");}
  default void blink()
    {out.println("I'm blinking");}
}


public interface TTail {
  default void shakeTail() {
    out.println("Wuush, I'm shaking my tail.");
  }
}
```

Then a new trait, `TCat`, puts together all the features previously defined, and a class implements it:

```
public interface TCat
    extends TEyes, TMouth, TTail {
  default public void purr()
    {out.println("PuUurRrRr");}
}


public class PersianCat implements TCat {
  private String name;
  public PersianCat(String n)
    {this.name = n;}

  public static void main(String[] args) {
    PersianCat jacky = new PersianCat("Jacky");
    jacky.eat("Meat");
  }
}
```

The output will be: `I'm eating Meat`.

Trait override. The override operator defines a new trait by adding one or more methods to an existing trait:

```
public interface TraitA {
  default void m()
    {out.println("I am m in TraitA");}
}


public interface TraitB extends TraitA {
  /** overrides TraitA, adding
      a new feature **/
```

```
  default void m2()
    {out.println("I am m2 in TraitB");}
}

public class C implements TraitB {
  public static void main(String[] args) {
    C c = new C();
    c.m();
    c.m2();
  }
}
```

Both methods `m` and `m2` are callable, therefore the output will be:

```
I am m in TraitA
I am m2 in TraitB
```

Trait override can be used to solve conflicts. If we add a method `close()` in both traits `TMouth` and `TEyes` (introduced above), we get from the compiler:

```
error: interface TCat inherits unrelated defaults
for close() from types TEyes and TMouth - public
interface TCat
```

In the overriding version of `close()`, we use the construct `X.super.m()`:

```
public interface TCat
    extends TEyes, TMouth, TTail {

  /** Conflict resolution **/
  default void close()
    {TEyes.super.close();}

  default public void purr()
    {out.println("PuUurRrRr");}
}
```

The method `close()` that will be executed is the one from the `TEyes` trait. The `close()` method from the `TMouth` trait is not lost, as it can be aliased. However, notice that the use of the `X.super.m()` feature reduces the low coupling between `TCat` and `TEyes`: if the `close()` method in `TEyes` changes (for example by adding a parameter to it), also the `close()` method inside `TCat` will have to change.

Exclusion. In [12], it was described the possibility to remove a default method by using the `default none` keyword, but this has not made its way in the official Java 8 release (dealing with negative information is never easy). A proposal for an exclude programming pattern, then, can use a well-know workaround, i.e., we can exclude a method by redefining it with an empty body or by throwing an exception. We prefer the second alternative. Consider this trait:

```
public interface TraitA {
  default void m()
    {out.println("I am m in TraitA");}
  default void q()
    {out.println("I am q in TraitA");}
}
```

If we want to exclude `m()`, we can do as follows:

```
public interface TraitB extends TraitA {
  default void m() {
    String s = "Method not understood";
    throw new UnsupportedOperationException(s);
  }
}

public class C implements TraitB {
  public static void main(String[] args){
    C c = new C();
    c.m();
    c.q();
  }
}
```

The first method call throws the exception. The second one would print `I am q in TraitA`.

Note that this programming pattern works well with respect to symmetric sum: if in all summed traits we have a method that we want to exclude, then this pattern will exclude simultaneously all upper method versions. However, we do not exclude the method for real, we just make unavoidable the upper implementation by overriding it, therefore Java introspection can still detect the excluded method: `C.class.getMethod('m')` still gets an answer.

Notice, however, that it is not possible to call in a new trait the excluded `TraitA` version of the method `m`:

```
public interface TraitB1 extends TraitB {
  /** It tries to rehabilitate the version
      from TraitA, excluded by TraitB **/
  default void m() {
    TraitA.super.m(); //does not compile
  }
}
```

If we try to compile the above code, we obtain an error: `not an enclosing class: TraitA - TraitA.super.m();`.

Alias. The alias operator provides another, alternative, name for referring to a certain method. Consider this trait:

```
public interface TraitA {
  default void mOneA()
    {out.println("I'm mOneA in A");}
  default void mTwoA()
    {out.println("I'm mTwoA in A");}
}
```

Now, in a new trait, we create an alias for `mTwoA()`:

```
public interface TraitB extends TraitA {
  /** Aliasing mTwoA() in
      aliasMTwoA() **/
  default void aliasMTwoA()
    {mTwoA();}
}

public class MyB implements TraitB {
  public static void main(String[] args) {
    MyB mc = new MyB();
    mc.aliasMTwoA();
    mc.mTwoA();
  }
}
```

The output will be:

```
I'm mTwoA in A
I'm mTwoA in A
```

When applying the alias programming pattern, attention must be paid to the alias name, as it is possible to override by mistake another method of the upper trait.

On the return type of methods. In Java, the name of a method is bound forever to its first introduction in terms of the return type. This impacts, in particular, on the re-introduction of a method name once this has been excluded, as our encoding of the exclusion operator relies on Java override. We discuss this issue by means of a pedagogical example. We want to develop a stack data structure (this example is taken from [6]). First of all, we show a single inheritance version:

```
public interface IStack {
  /* Tells if the stack is empty */
```

```
  public boolean isEmpty();
  /* Adds one item on the stack */
  public void push(Object obj);
  /* Removes and returns the first
     object on the stack */
  public Object pop();
}

public class Stack implements IStack {
  List<Object> l;

  public Stack()
    { l = new LinkedList<Object>(); }
  public boolean isEmpty()
    { return l.isEmpty(); }
  public void push(Object obj)
    { l.add(obj); }
  public Object pop() {
    if (!isEmpty())
      return l.remove(l.size()-1);
    else
      return null;
  }
}
```

Now, suppose that we must use another interface:

```
public interface IStackAlt {
  public boolean isEmpty();
  public void push(Object obj);
  /* Removes the first object on the stack */
  public void pop();
  /* Returns the first object on the stack
    (without removing it) */
  public Object getTop();
}
```

As we can see, this interface is different from `IStack` because of two methods: `pop()` is now `void`, and we have an additional method `getTop()`. We can implement this interface as follows:

```
public class StackAlt implements IStackAlt {
  List<Object> l;

  public StackAlt()
    { l = new LinkedList<Object>(); }
  public boolean isEmpty()
    { return l.isEmpty(); }
  public void push(Object obj)
    { l.add(obj); }
  public void pop()
    { if (!isEmpty()) l.remove(l.size()-1); }
  public Object getTop()
    { if (!isEmpty()) return l.get(l.size()-1);
      else return null; }
}
```

Notice that both methods `isEmpty()` and `push()` were already implemented inside the `Stack` class and we had to re-implement them inside the `StackAlt` class.

To switch to the trait-oriented approach, we introduce a `TStack` trait that defines all common operations:

```
public interface TStack {
  public List<Object> getStructure();

  default boolean isEmpty()
    { return getStructure().isEmpty(); }
  default void push(Object obj)
    { getStructure().add(obj); }
  default Object pop() {
    if (!isEmpty()) {
      int pos = getStructure().size()-1;
      Object o = getStructure().get(pos);
      getStructure().remove(pos);
      return o;
    }
    return null;
  }
}
```

Notice the abstract method `getStructure()`: it is a getter method to access the stack structure, that will be implemented as a field in a class, together with this method. The implementation of `TStack` is as follows:

```
public class Stack implements TStack {
  List<Object> l;
  public Stack()
    { l = new LinkedList<Object>(); }
  /* Glue Code */
  public List<Object> getStructure()
    { return l; }
}
```

Note that we put some glue code to provide the previously mentioned `getStructure()` method.

Now, we want to introduce a new method `getTop()` and we want to change the old `pop()` that was returning an `Object` into a `void` version. The first goal is easy, we can use the trait override pattern, while we encounter some problems with the `pop()` method:

```
public interface TStackAlt extends TStack {
  /** We redefine pop simulating
      the void return type **/
  default Object pop() {
    if (!isEmpty()) {
      int pos = getStructure().size()-1;
      getStructure().remove(pos);
    }
    return null;
  }

  /** We make the old pop still available
      (optional) **/
  default Object popTop() {
    return TStack.super.pop();
  }
  /** Trait Override **/
  default Object getTop() {
    if (!isEmpty()) {
      int pos = getStructure().size()-1;
      return getStructure().get(pos);
    }
    return null;
  }
}
```

We did provide an ad-hoc solution, by returning `null` in the new version of `pop()`. This is an implementing class:

```
public class StackAlt implements TStackAlt {
  List<Object> l;
  public StackAlt()
    { l = new LinkedList<Object>(); }
  /* Glue Code */
  public List<Object> getStructure()
    { return l; }
}
```

Notice that this solution preserves backward compatibility and it can be applied in similar cases. With respect to the single-inheritance version, the methods `isEmpty()` and `push()` are not duplicated anymore, the class tree is clearer, we provided a new `pop()` method with the new type but we also made the old one still accessible. Another successful case is when the type of the new version of the method is a subtype of the type of the old one, as

Java override is covariant. Any other cases involving uncomparable types force the break of backward compatibility.

5. Related work and conclusions

Traits as in [10] have been fully implemented in Smalltalk-Pharo [18]. A form of traits is present in PHP 5.4 [19]. The work [2] presents a version of traits with state (however, at the best of our knowledge, no satisfactory versions of stateful traits have been proposed so far). In [6] and [21] there are two proposals for traits in a Java-like language. The language XTRAITJ [3, 4] is a language for pure trait-based programming, providing complete compatibility and interoperability with the JAVA type system.

Traits and *mixins* are related. Both exploits composition instead of inheritance as a mechanism for software reuse and they are alternatives to multiple inherintance. Mixins [1, 5, 7, 8, 11, 22] are essentially subclasses parametric over their superclass, they can define fields and are a form of linearized multiple inheritance.

Aspect-oriented programming [15] shares with traits and mixins the goal of software reuse. However, their applications differ, as trait and mixin are for organizing the code, while aspects contain those parts of code that are cross-cutting concerns. While traits and mixins have a more general application, the code composition based on aspects is more fine-grained, as it is perfomed at the level of methods and not at the level of the containers of the methods.

In [20] there are two proposals to model a mixin-based style in Java 8, that is, a stateful approach. The first one exploits lambda expressions to model the state but suffers from some problems related to the runtime semantics of lambda expressions. The second proposal relies on the *virtual field pattern*, which is nothing else than the trait glue-code technique that we also exploit. However, this proposal does not consider the trait operators in detail.

This paper offers a view on how default methods can be exploited to promote and improve code modularization via an interface-as-trait programming approach. To this aim, Java 8 interfaces play, then, the role of traits, where abstract methods are the *required* methods (including the field accessor methods), and default methods are the *provided* methods. We have described some programming patterns inspired by the trait operators present in [10]: symmetric sum (to form a new trait by composing two or more existing traits), trait override (to form a new trait by adding methods to an existing trait), exclusion (to form a new trait by deleting a method from an existing trait), alias (to form a new trait to give a method an alternative name). The symmetric sum might introduce conflicts among method names, that must be solved with the use of trait override and exclusion.

As our interfaces-as-traits are stateless and accessor methods are the only (indirect) way to specify fields in traits, our approach imposes a restriction on visibility of fields. However, this is exactly how it works within stateless traits [10].

At the best of our knowledge, our proposal is the first one to explore the possibilty of a trait-oriented programming style in Java 8. A direction to explore is making it possible to exclude default methods (starting from [12], where it was described a `default none` keyword). Moreover, we believe our work could be also the base for reflecting about which form of traits (or even mixins) might be good as a language construct in future releases of Java.

It would be also interesting to refactor a large-scale, real-world example by applying our patterns and then use appropriate metrics (e.g., LOC) to measure the before- and after-factorization performances, in order to assess the degree of code modularity.

Acknowledgments

The authors would like to thank the anonymous referees.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. Jam — a smooth extension of Java with mixins. In *Proc. ECOOP '00*, volume 1850 of *LNCS*, pages 145–178. Springer-Verlag, 2000.

[2] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008. `http://dx.doi.org/10.1016/j.cl.2007.05.003`.

[3] L. Bettini and F. Damiani. Pure trait-based programming on the Java platform. In *PPPJ*. ACM, 2013. `http://doi.acm.org/10.1145/2500828.2500835`.

[4] L. Bettini and F. Damiani. Generic Traits for the Java Platform. In *PPPJ*. ACM, 2014. `http://dx.doi.org/10.1145/2647508.2647518`.

[5] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. ECOOP '99*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.

[6] V. Bono, F. Damiani, and E. Giachino. On traits and types in a Java-like setting. In G. Ausiello, J. Karhumki, G. Mauri, and C.-H. L. Ong, editors, *IFIP TCS*, volume 273 of *IFIP*, pages 367–382. Springer, 2008.

[7] V. Bono, J. Kusmierek, and M. Mulatero. Magda: A new language for modularity. In *ECOOP*, pages 560–588, 2012.

[8] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah, 1992.

[9] J. D. Darcy. JEP 126: Lambda Expressions & Virtual Extension Methods. `http://openjdk.java.net/jeps/126`.

[10] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 2:331–388, 2006.

[11] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269. Springer-Verlag, 1999. `http://dl.acm.org/citation.cfm?id=645580.658808`.

[12] B. Goetz. Interface evolution via virtual extensions methods. `http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf`, June 2011.

[13] B. Goetz and R. Field. Featherweight Defenders: A formal model for virtual extension methods in Java. `http://cr.openjdk.java.net/~briangoetz/lambda/featherweight-defenders.pdf`, March 2012.

[14] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[16] Lambda Expressions. Lambda Expressions for the Java Programming Language. `http://openjdk.java.net/projects/lambda/`.

[17] A. C. Oliver. Love and hate for Java 8. `http://m.javaworld.com/javaworld/jw-07-2013/130725-love-and-hate-for-java-8.html?mm_ref=https://www.google.it`.

[18] Pharo. `http://www.pharo-project.org/home`.

[19] PHP 5.4.0 Release Announcement. `http://php.net/releases/5_4_0.php`.

[20] F. Sarradin. Java 8: Now you have mixins? `http://kerflyn.wordpress.com/2012/07/09/java-8-now-you-have-mixins/`.

[21] C. Smith and S. Drossopoulou. Chai: Traits for Java-like Languages. In *Proc. ECOOP '05*, volume 3586 of *LNCS*, pages 453–478. Springer-Verlag, 2005.

[22] The Scala Group. Scala Website. `http://www.scala-lang.org/`.